

UNITED STATES PATENT APPLICATION
FOR
RETRIEVAL OF SERVICES BY ATTRIBUTE

INVENTORS:

JOHN HARVEY, III
ANDREW MICHAEL DANIELS
WILLIAM JAMES SAUNDERS

PREPARED BY:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN
12400 WILSHIRE BOULEVARD
SEVENTH FLOOR
LOS ANGELES, CA 90025-1026

(408) 720-8598

File No. 04860.P1637

"Express Mail" mailing label number TB230562563 US

Date of Deposit MAY 5, 1995

I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to the

Commissioner
of Patents and Trademarks, Washington, D.C. 20231.

JOANNE MILKY

(Typed or printed name of person mailing paper or fee)

Joanne Milky

(Signature of person mailing paper or fee)

RETRIEVAL OF SERVICES BY ATTRIBUTE

BACKGROUND OF THE INVENTION

1. Field of the Invention

5 The present invention relates to providing international services in computer systems. More specifically, the present invention relates to locating services, such as formatting, input and other services for providing international functionality in a computer system by the specification of attributes.

10 2. Background Information

As computer systems proliferate to beyond regional and national boundaries, demands for international software increase. To make software more international, services are made available which provide functionality within the system and application programs which allow the input and presentation of data which is
15 nationally, and even, culturally-specific. For example, services are provided which allow the input of text in a way which is understood by native speakers. Services may also include those which allowing the formatting of dates, monetary sums, provide sort ordering rules, and specify whether text is displayed using the roman or Cyrillic character set. These services may also include the encoding of the language,
20 and whether text in the language is displayed left-to-right, right-to-left, up-to-down, or any combinations of these. Other services may also be provided, according to implementation.

One of the ways in which international capabilities has been provided is through the use of locales. Locales are provided for a number of nationalities, and/or, cultures. Locales typically specify many of the above-services, including text services, sorting services, formatting services, spell-checkers, and other services
5 which require culturally-specific facilities. Typically, a large number of these services are provided under a locale label, and provide the services as a single monolithic group of services which is difficult to modify or "localize" to a particular region, culture, and/or individual or group of individuals.

One shortcoming in certain systems implementing locales, or other
10 international functionality, is that they frequently suffer from a very small naming space in relation to the number and variety of services which may required by future application programs, including, those requiring international, regional, or other localized services. On the Macintosh brand computer system, the for example, Script Manager, International utilities Package, WorldScript I, WorldScript II, the
15 Language Manager, and the Text Services Manager work together to provide international functionality. The first four organize data based on Script ID. On the Macintosh a Script represents a writing system. Scripts also indirectly define a character encoding. Script specific data is named by assigning ranges of possible resource IDs to each script. Thus to determine which script a certain data type
20 works with, the data type's resource ID is retrieved and the script ID can be calculated back from the resource ID. This system does not provide an explicit way to categorize data by language, encoding, region, or other attributes which an application program may require. An application or system service that wants to do

this must develop some method of its own. This limited name space results in data that is not really writing system specific being forced into a categorization by script code.

Thus, due to the proliferation of computer systems, especially across national
5 and regional boundaries, new methods which explicitly categorize services by
language, encoding, region, etc..., is required which addresses many of the
shortcomings of the prior art. This service should provide an ample naming space
for any additional functionality to be provided, and provide means for accessing
those services irrespective of a locale, but rather, based upon some other criteria.

SUMMARY

An automatic method and apparatus for providing services on a computer system. A set of services is established providing different functionality for the computer system. With each of the set of services an attribute is stored for
5 identifying characteristics of the function of each of the set of services. Then, the first set of services can be retrieved by a first set of attributes which is specified irrespective of any locales which may be defined in the computer system. Thus, characteristics or attributes of most importance to a service may be used to describe it and used for retrieval of such services.

10 The services may include text services, formatting services, sorting services, etc., and may be used for providing international functionality. The establishing of the first set of services includes building a structure which references each of the first set of services by the attributes. Retrieving may include maintaining a reference to one of the services in the structure and iterating through the structure in order to
15 retrieve a subsequent one of the services. An iterator may be created which is used to iterate through the database of services and provide the services to application or system programs.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example and not limitation in the figures of the accompanying in which like references indicate similar elements and in which:

5 Figure 1^{Shows} how Locale Files, Text Services, and Localization files map to Locale objects in a Locale Object database in embodiments of the present invention.

 Figures 2a and 2b show an example system in which an embodiment of the present invention may be implemented, including, the interaction between locales, localizations, the locale object manager, the locale database and application
10 programs.

 Figure 3 shows the organization of a locale lookup list, text services, locales and localizations.

 Figure 4 shows an example of the locale lookup list.

 Figure 5 shows a format of a locale file and the references to locale objects.

15 Figure 6 shows an example locale file and the corresponding references to locale objects.

 Figure 7 shows a format of a locale object and the references to attributes and name subtables, and to the data referenced by the locale object.

 Figure 8 shows an example of a locale object.

20 Figure 9 shows the process performed during initialization of the locale object manager.

Figure 10 shows a plurality of processes resident in system memory which provide the application program interface (API) to application programs requiring locales and localization services.

Figure 11 shows a process for creating an iterator to iterate through the locale
5 database in order to obtain services requested by an application.

Figure 12 shows a datum which may be used for storing characteristics of an iterator in the Locale object manager.

Figure 13 shows a process for iterating through the locale database to obtain services requested by an application.

10 Figure 14 shows a sub-process for locating the next object in the Locale database which has a matching attribute.

Figure 15 shows the locale database resident in the locale object manager's system memory, a plurality of iterators in the database, and the results of iterating through the locale database.

DETAILED DESCRIPTION

The present invention relates to providing international services in a computer system, such as those used for inputting and presenting text to a user in a computer system. Embodiments of this invention has many applications, however, it may be particularly useful for providing different services in a computer system
5 irrespective of locale. These allow the retrieval of services bases on characteristics or attributes of the objects which are most important to requesting processes.

Although the present invention will be described with reference to certain specific embodiments thereof, especially, with relation to certain hardware configurations,
10 data structures, packets, method steps, and other specific details, these should not be viewed as limiting the present invention. Various modifications and other changes may be made by one skilled in the art, without departing from the overall spirit and scope of the present invention.

A portion of the disclosure of this patent document contains material which
15 is subject to copyright protection. They copyright owner has no objection to the facsimile reproduction by anyone of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever. Copyright © Apple Computer, Inc.

Implemented embodiments of the present invention de-couple the
20 relationships between locales and services provided for specific locales. The resource type and resource ID limitations which existed in the prior art Macintosh brand environment, and its associated limited name space, has been addressed by implemented embodiments of the present invention. In this paradigm, objects such

as 101, 102, and 103, which provide a variety of services, including, but not limited to, date formatting services, sorting order services, and other text services, such as input methods and tokenizations. Thus, an input method which may have typically been associated with a particular Locale, and by extension, the script in which that locale is defined, may be used irrespective of identifiers such as resource type and resource ID values within a specific range for the script.

A new process resident in computer system memory 204, known as the Locale object Manager, and which is executed by the computer system's processor, as will be described below, provides such cross-Locale services to provide different functionality. Thus, instead of associating services or objects such as 101-103 with particular Locales such as 110-111, any Locale may reference any service available to the Locale object manager. In addition, localizations 120 may be defined, which may reference Locales 110 and 111 and Locale objects, such as 101-103, in order to localize the computer system in which the Locale object Manager is active.

Because characteristics of the services are stored, and the characteristics can be any characteristic of a service, irrespective of language or locale, the system is much more flexible than certain prior art systems. Rather than a finite number of different locales and certain specific services within those locales, configuration of a system can be made much more specific than those using prior art locales and localizations, even allowing the configuration of the system according to a particular user's requirements. This is because the services are stored with associated attributes which define the services.

A typical system configuration in which embodiments of the present invention may be implemented is shown in Figure 2a. In various embodiments of the present invention, a general purpose computer system is used for implementing the processes, objects and associated services to be described here. A computer system, such as a workstation, personal computer or other processing apparatus 200 is illustrated in more detail in Figure 2a. 200 comprises a bus or other communication means 201 for communicating information, and a processor 202 coupled with bus 201 for processing information. System 200 further comprises a random access memory (RAM) or other volatile storage device 204 (referred to as main memory), coupled to bus 201 for storing information and instructions to be executed by processor 202. main memory 204 also may be used for storing temporary variables or other intermediate information during execution of instructions by processor 202. Included in memory 204 during run-time may be the conference component module which operates according to the communication protocols to be described below. System 200 also comprises a read only memory (ROM) and/or other static storage device 206 coupled to bus 201 for storing static information and instructions for processor 202, and a data storage device 207 such as a magnetic disk or optical disk and its corresponding disk drive. Data storage device 207 is coupled to bus 201 for storing information and instructions.

System 200 may further be coupled to a display device adapter 221 such as a cathode ray tube (CRT) or liquid crystal display (LCD) coupled to bus 201 for displaying information to a computer user. Such a display 221 may further be coupled to bus 201 for the receipt of video or image information. An alphanumeric

input device 222, including alphanumeric and other keys may also be coupled to bus 201 for communicating information and command selections to processor 202. An additional user input device is cursor control 223, such as a mouse, a trackball, stylus, or cursor direction keys, coupled to bus 201 for communicating direction
5 information and command selections to processor 202, and for controlling cursor movement on display 221. A hard copy output device 227 can also be coupled to bus 301 for generating hard copy information, including that including international data such as multi-locale information using multiple objects. Embodiments of the present invention may be especially useful for inputting international information to
10 application programs from input devices 222 and 223, and output of information on devices 221 and 227 in a culturally-specific, or even, user-specific manner. More specific services can be located and used using embodiments of the present embodiments of the present invention, rather than certain prior art systems which were limited by the pre-designated locales and localizations.

15 Note, also, that any or all of the components of system 200 and associated hardware may be used in various embodiments, however, it can be appreciated that any configuration of the system may be used for various purposes according to the particular implementation.

In one embodiment, system 200 is one of the Apple Computer® brand
20 family of personal computers such as the Macintosh 8100 brand personal computer manufactured by Apple Computer, Inc. of Cupertino, California. Processor 202 may be one of the PowerPC brand microprocessors manufactured by Motorola, Inc. of Schaumburg, Illinois.

Note that the following discussion of various embodiments discussed herein will refer specifically to a series of routines which are generated in a high-level programming language (e.g., the C or C++ programming language) and compiled, linked, and then run as object code in system 200 during run-time within main
5 memory 204 by processor 202. For example the object code may be generated by the C++ Compiler available from Symantec, Inc. of Cupertino, California.

Although a general purpose computer system has been described, it can be appreciated by one skilled in the art, however, that the following methods and apparatus may be implemented in special purpose hardware devices, such as discrete
10 logic devices, large scale integrated circuits (LSI's), application-specific integrated circuits (ASIC's), or other specialized hardware. The description here has equal application to apparatus having similar function.

Figure 2b shows a more detailed view of the files resident in the mass storage device 207 and main memory 204. Locale information 212 is loaded from
15 mass storage information 212 by Locale object Manager 214 across bus 201 by processor 202, and is used to reference locale objects 211. Locale information 212 and Locale objects 211 are used to build the Locale object database 213 in main memory 204 of the computer system during system boot-time. Localization files 210 may also be read and used to modify the locale database according to any
20 defined localizations. These may reference both Locales defined by Locale information 212 and Locale objects 211. The Locale database 213 may be file-mapped or may be structured to allow rapid searching by name and by attribute(s).

The Locale Object manager 214 scans files 211, 212, and 214 at boot time and uses the data inside these files to build the Locale database 213. The types of files that are scanned for, and where to look for these files is determined by the Locale Object file lookup list('lfl') resource 301 shown in Figure 3. This resource
5 exists in either a system file, in a Locale Object Manager's preference file, or some other pre designated location.

File types that contain information to be included in the Locale Object Manager's database 213 are required to contain either a Locale Object directory resource ('ldir') and/or a collection of locale object resources ('lobj'). Locale Object
10 directory resources specify what kinds of data a file contains and provides enough information for the Locale Object manager to find the data types.

Locale Object resources describe a data object itself. Data which is intended to be catalogued by the Locale Object Manager is accompanied in embodiments of the present invention by a name table and an attributes table. The Locale Object
15 Manager uses the information in these tables to create keys for finding the data. This allows the specification of name and attributes at system runtime to retrieve services irrespective of locale.

When the Locale object Manager 214 is first initialized it loads the 'lfl' list file 301 and proceeds to lookup file types described in the list sequentially. The first
20 file type described in the list is assigned to Locale Files which have the file type 'lcfl'. The Locale file Lookup list references a Locale directory 320 which may contain a plurality of Locale files 321. After the entry for Locale files other entries can have any order.

The format of the Locale file lookup list resource is a list of FileLookupEntry structures preceded by a long word count.

Table 1. Locale File Lookup List Entry

OSType	FileType
OSType	Folder to search

5

Table 2. Locale File Lookup List

UInt32	Count
FileLookupEntry	Entries[Count]

Figure 4 illustrates how a 'lfl' resource 400 would map to actual files. The 'lfl' in the figure lists 3 file types, as indicated by the count field 401. The first is the type 402 for Locale Files ('lcl') which are located in the Locale Folder ('loca'). This directory is specified in field 405. The second file type in the 'lfl' file 400 is 'lclz' field 403 which specified the file type for Localization files. These are also located in the Locale directory 420, as referenced by field 406. The Localization files 430 may actually be in a subdirectory, shown as "Localizations", however, the Locale object manager 214 searches all subdirectories in a specified directory. The last item in the 'lfl' is the type and directory location for Text Service files. These may include text input methods like the Japanese Input Method Kotoeri. As shown

in the figure, these may be located in a pre designated directory entitled "Extensions" 410.

5 In the prior art Macintosh brand System versions 7.x, an international configuration resource having the resource type 'itlc' is defined which specifies which script and geographical region the system is localized for. Locale Object Manager 214 uses the system script field in the 'itlc' to set the default current Locale, which is retained in memory in the Locale object manager 214. The 'itlc' is also entered into the Locale database. It can be retrieved by querying the Locale database for Locale objects with the name 'itlc'.

10 All other resources that the Locale Object manager 214 reads are directory based. Each resource begins with a list of records that provide the offset, length, a checksum that can be used for verification, and an identifying 4-byte tag. Directory entries are described in Table 3. Table 4 illustrates the structure of a directory.

Table 3: Locale Object Directory Entry

OSType	TableTag	Identifies table type
UInt32	checksum	can be used to verify data integrity
UInt32	offset	offset from beginning of the resource to the actual table
UInt32	length	length in bytes of the table.

Table 4: Directory

UInt32	count	Number of Locale Object Directory Entries in Table
LocaleObjectDirectoryEntry	Entries[count]	Array of LocaleObjectDirectoryEntries

5

Locale Files have the format as shown in Figure 5. They contain a resource type called a Locale Object directory ('ldir') 501. Directory resources such as 501 contain 2 subtables: the Locale object list subtable 503 and the Locale header subtable 502. Locale directories may optionally contain other subtables. The

10 Locale object list subtable 503 references other Locale objects which defined the Locale objects themselves. A name table (discussed below) is added that includes manufacturer strings, copyright information, and other application visible names, which may be used by the application and/or the Locale object manager 214.

The Locale Header subtable 301 contains three values: A 32-bit word

15 indicating the format, a 32-bit word indicating the Script ID this Locale maps to, and a 32-bit word which contains bit flags characterizing this data. A header subtable is identified by the tag 'head'.

Table 5: Locale Header

UInt32	format	Version number (e.g. 0x00010000)
UInt32	ScriptID	If ScriptID is not relevant set to -1
UInt32	flags	bit flags

Different flag values can be contained in the flags field of the header to specify special treatment of the locale object. Two of these include kNeedsFSObjectSpecification, and kLoadObject. kNeedsFSObjectSpecification specifies that the Locale Object Manager should create a FSObjectSpecification and associate it with the entry created in the Locale Object database for a given resource. Clients such as the Language Manager in the Macintosh brand computer system can use this bit to insure that files containing resources can be located without requiring a search through the file system. The kLoadObject flag indicates that the Locale object should be loaded into physical RAM.

The Locale object list subtable 503 lists all the Locale objects associated with a Locale and how and where to find them. The subtable format is described in Tables 6 and 7. A Locale object List subtable 503 is identified by the tag, 'lobl'.

15 **Table 6: Locale object List Entry**

OSType	dataDescriptor	describes the data reference
UInt32	offsetOrID	If data is a resource this is the resource ID. All resources have a type of 'lobj'. If data exists in a data fork this is an absolute offset to the start of the data.
UInt32	dataLength	length of the data. Can be 0 for a resource reference.
OSType	externalFileLocater	If this object is inherited this field contains the resource type of the resource that contains the file locator. There can be three types: alis -- file alias, prfo -- a scarecrow permanent file reference, and 'none' -- used when a the object does not inherit.
UInt32	externalFileResID	Resource ID of the resource containing the file locator.

Table 7: Locale Directory Subtable

UInt32	format	format descriptor, should be 0x00010000
UInt32	count	Number of LocaleObjectLocators to follow.
LocaleObjectLocators	objectList[count]	Array of object locators

The Locale Directory subtable is used to allow inheritance. This allows the specification of a service which actually resides in another locale file, avoiding the need to duplicate the information.

For each Locale Object file that the Locale Object Manager 214 finds, it opens it and locates a locale directory resource. A locale directory resource contains a subtable that describes the locale objects that make up a locale and how to find those objects. For each entry in the locator subtable the Locale Object Manager 214 will attempt to find the described locale object. If the described locale object is successfully located then the Locale Object Manager 214 will create an entry in the Locale database 213. The 'ldir' resource, such as 501, can be viewed as the root of one of the trees, and all the associated Locale objects as leaves.

Figure 6 illustrates an example Locale Directory resource 600. It contains a count field 606 which specifies that the Locale Directory resource references two Locale objects. It also contains a header field 602 per the format shown in table 3 above. The directory resource 600 further comprises a Locale object directory subtable 602, a header subtable 603, and a Locale object list subtable comprising Locale object list entries 604 and 605. Locale object list entries 604 and 605 have the type 'rsrc' because the referenced Locale objects 610 and 620 are resources. These Locale objects have resource ID's 0 and 12 as shown in the illustration. The actual content of a Locale object, such as Locale objects 610 and 620 which are

referenced by these entries in the figure, is illustrated below in Figure 8. The Locale objects 610 and 620 referenced by the object list entries both exist in the same file as the Locale Directory resource 600. This is indicated by the 'none' constant in the externalFileLocator field of the Locale object list entries 604 and 605.

5 Locale object Manager 214 creates database entries for each Locale object found in the files it examines at boot time. For Locale files the Locale directory resource, such as 600, is used to locate Locale objects. For other file types in the Locale file lookup list the Locale Manager iterates through the locale objects contained in the files resource fork and creates database entries for each one that is
10 found.

 Locale object resources have the format 700 shown in Figure 7. It can contain the actual data or information that a client can use to load the data from somewhere else. This type of data is encapsulated in the 'data' subtable 702, which references the data 705. The information contained in the 'data' subtable 702 is
15 unknown to the Locale object manager 214. It retrieves a reference to the object 705 upon an appropriate call from a client. Beyond its offset and length in the 'lobj' resource, such as 604 and 605, the Locale object manager 214 knows nothing about subtables contained within data 702.

 Locale objects such as 700 also contain a names subtable ('name') 703. The
20 names subtable 703 is used to associate a collection of text names with an object or file. Names in name table 706 can be used for user visible text or as keys in the locale database. Names in name table 706 can be used, for example, for specifying the function performed by an object (e.g. "SORT"). If a name in the names table

706 is intended to be seen by a user it could specify encoding and language information. If the name string is not intended to be seen by a user than the encoding and language information is set to -1.

Objects in the Locale database 213 are found by the Object Manager by
5 name and attribute(s). The name of an object is taken from names table 706. The attributes associated with an object are taken from the attributes table 704 which is referenced the attributes subtable 701 contained within the object.

An attribute is a named value. The name is a string preceded by a length
byte. The value of an attribute is a 16-bit word which indicates the byte length of
10 the value followed by the actual data. An attributes table is a collection of name value pairs preceded by a 32-bit word which contains the number of name-value pairs in the table.

Tables 8-11 describe the data structures that go together to create a locale object resource.

15

Table 8: Locale Name Record

UInt32	BaseEncoding	Implementation-dependant code indicating the Base Character Encoding in
UInt16	variantEncoding	Implementation-dependant code indicating the variant Character Encoding
UInt16	charFormat	Implementation-dependant code indicating the variant Character Encoding
UInt16	languageID	Language Code
UInt16	nameID	name identifier
UInt8	nameLength	length of name
UInt8	name[nameLength]	The name.

In implemented embodiments, some possible names available for the nameID may include: kKeyNameIndex, kUserName, kCopyright, kManufacturer,

kFunctionDescription, and kVersion. Other nameID's may also be used, according to implementation. Using NameID's allows an object to contain multiple localized name strings.

5 **Table 9: Names SubTable**

UInt32	format	Format of subtable (0x00010000)
UInt32	count	count of NameLocaleRecords
LocaleNameRecord	nameRecords[count]	Array of NameLocaleRecords

Table 10: Attribute Record

UInt8	nameLength	length of attribute name.
UInt8	name[nameLength]	attribute name
UInt16	valueLength	length of value
UInt8	value[valueLength]	value data

Table 11: Locale object

Directory	Directory of subtables
Data	data subtable
Names	Names subtable
Attributes	Attributes subtable

10

Table 11 lists the required subtables that a Locale object contains.

Additional subtables can be added by simply adding the subtable and creating a new entry in the Directory table in the Locale object. There is no order requirement except that the directory table must be the first subtable in the resource. Locale objects that should be loaded into memory include a header table that contains a flag field with the kLoadObject flag set.

15

Figure 8 illustrates an example of a Locale object resource 800 and its subtables. The directory contains three entries 802, 803 and 804, preceded by the

directory version and count fields 801 indicating the number of entries in the directory. The Locale object 800 illustrated contains a 'kcs4' resource in its data table 820, a name subtable 830 containing two Locale name records 831 and 832 (specifying a key name and user name), and an attribute subtable 810 containing two
5 name/value pairs (attributes) 811 and 812. The first attribute or name/value pair 811 specifies that the object is characterized as having an attribute named 'rezid' with a value of 1. The second attribute 812 specifies that this object also has an attribute named 'script' which contains a value of 0.

The flow of Locale data from Locale lookup list files into the Locale
10 database 213 as performed by the Locale Object manager 214 is illustrated in Figure 9. In these implementations, this process occurs at system boot time, but it may be performed at any implementation-dependent interval.

At system boot time or other implementation-dependant interval, the Locale object manager 214 loads the Locale Lookup list file resource at step 902. The
15 Locale object manager then scans the system's mass storage device 207 or other file system for the presence of all the file types specified in the Locale Lookup list file at step 904. For each file, it is determined whether the file is a Locale file at step 906. If not, then the Locale objects are retrieved by iterating through all the Locale object resources contained in the file at step 910. These are then used to create entries for
20 the Locale objects at step 909 to build the Locale database 213. The Locale database 213 may be either file-mapped, or stored in an implementation-dependant format to speed searching.

In the case of Locale files, as detected at step 906, the associated Locale directory resource is then retrieved at step 907. Using the information contained in the Locale directory resource, the Locale objects are then retrieved at step 908 and stored into the Locale database 213 at step 909.

5

Application Program Interface (API)

Clients request that the Locale object Manager 214 search the database by passing names and a collection of attributes (named values) via its Application Program Interface (API). The supported API calls to procedures resident within the Locale object Manager 214 during computer system runtime are illustrated as 1001-1019 in figure 10. Searches can be done with an iterator. An iterator is created by the API call LocaleIteratorCreate 1005. In the call to LocaleIteratorCreate a name and collection of name-value pairs can be passed. Subsequent to the creation of the iterator, LocaleIterate is called to retrieve searches. Optionally, a client can search for a single object with the API call SearchOneLocaleObject 1011. The client has the option of receiving the data immediately as a result of the search, retrieving a reference to the Locale object (LocaleObjectRef) or both. LocaleObjectRefs allow a client to delay the retrieval of the actual data until later, when the object is actually required. A summary of the API calls used in implemented embodiments follows.

20 GetCurrentProcessLocaleRef 1001 - A client calls this function to obtain the default LocaleRef for the current context. A default LocaleRef is retained in the Locale object Manager.

 GetLocaleRef 1003 - Called to find the Locale that maps to a script code.

SetCurrentProcessLocale 1003 - Called by a client to change the Locale of the current context.

LocaleIteratorCreate 1005 - Creates an iterator object that can then be used to iterate through the Locale database 213. The caller passes the direction to iterate in, for the Locale database that can only be forward or backward, the name of the type of object to look for, a pointer to 0 or more NameValuePairs, and the number of NameValuePairs.

LocaleIteratorDispose 1006 - Disposes of a LocaleIteratorRef.

LocaleIterate 1007 - Called to iterate through the Locale data. The search will return a pointer to the objects data in the objectData parameter if a valid object is found. ObjectRef is an opaque reference to the object itself. When there are no more objects left in the database that correspond to the name and attributes specified then the error eObjectNotFound is returned. LocaleIterators can be saved and reused with different names and attributes. However, before beginning a new search a caller should be sure to reset the iterators start point by calling SetLocaleIterator. A pointer to the object's data is returned in the objectData parameter, and the size of that data is returned in a dataSize parameter. A reference to the entire Locale object is returned in an objectRef parameter. If a caller is not interested in any of these three items NULL can be passed. Data and name information from the referenced locale object can be extracted via the functions GetLocaleObjectData and GetLocaleObjectName. These functions are described below.

GetLocaleObjectData 1017 - Used to extract data from a LocaleObjectRef which was found via LocaleIterate. The data pointer is a pointer into global read-only memory.

5 GetLocaleObjectName 1016 - Retrieves the name(s) of the object contained in the names subtable.

SetLocaleIterator 1009 - Change where the iteration will begin. If the locale parameter is NULL the iterator will reset to its original starting point.

SearchOneLocaleObject 1011 - This is used for a direct search. The first object matching the name and attributes is returned. No iterator is required.
10 Internally, this routine is equivalent to calling LocaleIterator with a NULL iterator object, and then stopping the search as soon as one object is found.

AddLocaleObject 1013 - Adds an object to a given Locale.

RemoveLocaleObject 1015 - Removes an object from the Locale database
213. This routine does not dispose the actual data.

15 GetObjectNameAsTextObject 1016 - Uses the nameID and encoding specification to find a name, and returns that name as a TextObject.

GetLocaleObjectAssociatedData 1017 - Once an object has been obtained through a database search the caller can query to see if other associated data is attached to that object. When objects have the create FSObjectSpecification flag set
20 in the flags field of the header subtable, the Locale Manager will create an FSObjectSpecification and associate it with the object. Clients that need to subsequently open the file such as the Language Manager or the Text Services manager in the Macintosh brand system use this routine to obtain an

FSObjectSpecification. This is done by calling the routine with a tag of 'fsos'. Static data such as the class and function name or the language for which a file is localized is stored in the data subtable.

AssociateDataWithLocaleObject 1019 - Associates the data pointed to by a
5 dataPtr with the Locale object.

A LocaleIteratorReference is created to iterate through the data contained in the Locale object Manager's database 213. Multiple iterators, each with different specified attributes (or no attributes at all, if desired, to simply retrieve the next locale object), can be used by the same or different processes for iterating through
10 the database and retrieving different objects. Thus, one iterator may be used for retrieving sorting services, and another may be used for retrieving date formats. As already discussed, the iterator must first be created, and then, the client can cause the iterator to iterate through the database to retrieve the next object having the specified attributes, if any.

15 NameValuePair structures are filled out by clients prior to an API call. They are the means by which attributes that a client is interested in are passed to the Locale Manager's search engine. Clients ask the Locale Manager to find data using name and attributes. Attributes are named values. Attribute names can be any ASCII string, but there are a number of predefined attribute names in implemented
20 embodiments, as follows: kScriptName, kLanguageName, kRegionName, kResIDName, kEncodingName, and kTextServiceTypeName. Other attribute names may be used, according to implementation, with different associated values to obtain

the desired object. Each of these may be passed in an API call to Locale object Manager 214, and the Locale object may be returned responsive thereto.

A summary of the process used for creating an iterator is shown in Figure 12. Parameters passed to the process include a LocaleRef, the direction to iterate in, the
5 name of the object to look for (if desired), a pointer to one or name/value pairs (if desired), and a count of a number of name/value pairs passed. If no name or name/value pairs are passed, then the iterator is created at the specified LocaleRef. Calls to LocaleIterate result in every object in the database being returned on each call. If no LocaleRef is specified, then iterator is set to the default or current Locale,
10 depending upon the implementation.

LocaleIteratorCreate process 1005 starts at step 1102 wherein an iterator is created to be used as a reference for subsequent API calls. At step 1106, wherein it is determined whether a LocaleRef has been specified. If so, then the current reference for the iterator is set to the specified locale at step 1108. If not, the
15 reference is set to the current locale which is retained within the Locale object manager 214 at step 1110. The iterator reference is then returned from process 1005.

Figure 12 illustrates an iterator data structure 1200 which may be used to retain the state of the iterator. The structure includes a current locale reference 1201.
20 The structure also includes the specified name 1202, if any. In addition, the structure can include specified attribute(s) 1203, if any. Finally, a last object flag 1205 is also retained. Thus, upon a subsequent call to LocaleIterate, the function can immediately exit returning a value eObjectNotFound.

Figure 13 illustrates a detail of the steps taken during iterations to return an object from the Locale database 213. The process starts at step 1302 wherein it is first determined whether the iterator has been created. If not, then the function aborts indicating that the iterator was never created. If, however, the iterator was created, the last object flag is examined at step 1306. If it is set, then the process returns indicating eObjectNotFound. If the last object flag was not set, as detected at step 1306, then the process proceeds to step 1310 wherein the next object having the specified name and attribute(s). This is to prepare the iterator for the next call to LocaleIterate. It is again determined, at step 1312, whether this object was the last object having the name and the attribute(s). If so, then the last object flag is set at step 1314.

If there are more objects in the database having the name and attribute(s), the process determines at step 1316 whether this is the first pass through the process. If so, then the current reference is set equal to the next reference at step 1318, that is, the next object reference retrieved at step 1310. This is so that on subsequent calls to LocaleIterate, that process has already pre-iterated to the current object requested on a call. The process can abort immediately after entry if there are no additional objects (e.g. as detected at step 1306). In the event of a first pass, process 1007 then repeats step 1310. Upon the completion of process 1007, the current object is returned to the client, along with the iterator reference.

Figure 14 shows in more detail the steps which may be taken for locating a next matching object within process LocaleIterate 1007. It is first determined if there are any more matching objects in the current locale at step 1402. This step

may be performed in a database 213 which arranges objects by Locale. If not, then the process proceeds to step 1404 to determine whether there are any additional defined Locales within the database which have matching objects. If so, then at step 1406, the next Locale is retrieved, and the matching object is retrieved at step 1408.

- 5 The reference to the matching object can then be returned. If not, then an eObjectNotFound message may be returned to the may process at step 1407.

As shown by Figure 14, Locale boundaries can be crossed when iterating through the database with LocaleIterate. Specifying a LocaleRef when calling LocaleIteratorCreate describes where a search begins. However, when searching
10 with an iterator all objects in the database are examined. In short, a Locale object belonging to a different Locale than the one which was passed to LocaleIteratorCreate can be returned as the result of LocaleIterate.

This is further illustrated with reference to Figure 15. For example, if a Locale Database 213 has the structure shown in Figure 15, wherein defined Locales
15 such as 1510, 1530, and 1550 hierarchically reference all names which are valid within the Locale. For example, a structure defining a Locale 1510 references the application visible names "Sort" 1511, "Script" 1512, and "Date Formats" 1513. Referenced by a name, such as "Sort" 1511, could be objects 1514, and 1515, which each have unique name/value pairs specifying that the encoding for the sort objects
20 is ASCII and Unicode, respectively. If an iterator is currently referencing the Unicode Sort object 1515, then an iteration may cause the iterator to then reference the next object, ASCII object 1511. Another iteration may cause the iterator to then cross the locale boundary 1519 between the German Locale 1510 and the US

English Locale 1530 to reference one of the sorting objects 1534 or 1535. In the case that the attribute specifies an ASCII encoding sort object, the first iteration out of the German locale object 1514 will immediately cause LocaleIterate process 1007 to return a reference to object 1534, crossing the locale boundary 1519. Any of the
5 locale boundaries 1519, 1539, and 1559 can be crossed by calls to LocaleIterate, even by the same application program.

As previously described, a single application can have multiple iterators for retrieving multiple services/objects. As shown in Figure 15, which may be the application-visible context of the database for a single application program, an
10 iterator such as the second LocaleIterator 1502 which may be used for retrieving date formatting objects. Note that the iterator is not restrained, like iterator 1501, from traversing across locale boundaries. Thus, a single application may reference multiple iterators each which reside in different locales, or are typically associated with different Locales.

15 Thus, by use of the foregoing, improved capabilities from different services, irrespective of locale, can be provided to application programs. Objects or services provided by such objects may be retrieved based upon relevant attributes, irrespective of which Locale or locale, as in the prior art, the objects have been defined for. It will be appreciated that though the foregoing has been described
20 especially with reference to Figures 1-16, that many modifications made be made, by one skilled in the art, without departing from the scope of the invention as described here. The invention is thus to be viewed as limited only by the appended claims which follow.